IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant	: Gilbert Wolrich et al.	Art Unit	: 2154
Serial No.	: 10/069,229	Examiner	: Joshua Joo
Filed	: December 11, 2002	Conf. No.	: 1429

Title : BRANCH INSTRUCTION FOR PROCESSOR ARCHITECTURE

MAIL STOP APPEAL BRIEF – PATENTS
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

APPEAL BRIEF ON BEHALF OF
GILBERT WOLRICH, MATTHEW J. ADILETTA, WILLIAM R. WHEELER,
DEBRA BERNSTEIN, AND DONALD F. HOOPER

The fees in the amount of $500 are being paid concurrently on the Electronic Filing System (EFS) by way of Deposit Account authorization. Please apply any other required fees to deposit account 06-1050, referencing the attorney docket number shown above.

(i.) Real Party In Interest

The real party in interest in the above application is Intel Corporation.

(ii.) Related Appeals and Interferences

The Appellant is not aware of any appeals or interferences related to the above-identified patent application.

(iii.) Status of Claims

This is an appeal from the decision of the Primary Examiner in an Office Action dated March 17, 2006, rejecting claims 1-21, all of the claims of the above application. The claims have been twice rejected. Claims 1-21 are the subject of this appeal.

(iv.) Status of Amendments

All amendments have been entered. Appellant has filed herewith a Notice of Appeal on **July 14, 2006**.

(v.) Summary of Claimed Subject Matter

Background

The claimed invention relates to branch instruction in which an instruction stream may execute in sequence and branch from the sequence to a different sequence of instructions. [Specification page 1, lines 9-12]

Appellant's Invention

Claim 1

One aspect of Appellant's invention is set out in claim 1, as a method for operating a processor. "Referring to FIG. 1, a communication system 10 includes a processor 12." [Specification, page 1, lines 2-5]

An inventive feature of Appellant's claim 1 includes executing a branch instruction that causes a processor to branch from executing a first sequential series of instructions to a different sequential series of instructions based on a byte, specified by the branch instruction, in a register,

specified by the branch instruction, being equal or not equal to a byte value specified by the branch instruction. "Byte_spec Number specifies a byte in register to be compared with byte_compare_value. Valid byte_spec values are 0 through 3. A value of 0 refers to the rightmost byte. A byte_compare_value is a value used for comparison. Valid byte_compare_values are 0 to 255. The label# is a symbolic label corresponding to the address of an instruction." [Specification, page 13, lines 6-21] Appellant's FIG. 5 shows exemplary formats for branch instructions. [FIG. 5]

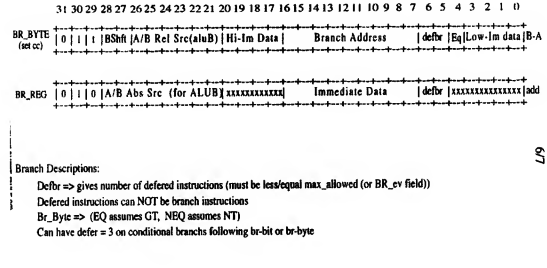

FIG. 5

"BR=BYTE, BR!=BYTE. This branch instruction branches to an instruction at the specified label if a specified byte in a longword matches or mismatches the byte_compare_value. The br=byte instruction prefetches the instruction for the "branch taken" condition rather than the next sequential instruction. The br!=byte instruction prefetches the next sequential instruction. These instructions set the condition codes in the microengine.

Format: br=byte[reg, byte_spec, byte_compare_value, label#], optional_token

br!=byte[reg, byte_spec, byte_compare_value, label#], optional_token

Reg A is a context-relative transfer register or general-purpose register that holds the operand." [Specification, page 13, lines 6-21]

Claim 14

Claim 14 recites another aspect of the invention. Claim 14 is a computer program product residing on a computer readable medium comprising instructions, including a branch instruction. "Computer code whether executed in parallel processing, pipelined or sequential processing machines involves branches in which an instruction stream may execute in a sequence and branch from the sequence to a different sequence of instructions." [Specification, lines 9-12]

Inventive features of Appellant's claim 14 include the branch operation causing a processor to fetch a byte, specified by the branch instruction, stored in a register, specified by the branch instruction. "This branch instruction branches to an instruction at the specified label if a specified byte in a longword matches or mismatches the byte_compare_value." [Specification, page 13, lines 6-7] "Example: br!=byte[reg, byte_spec, byte_compare_value, label#], defer[3]. This instruction represents an instruction that compares an aligned byte of a register operand to an immediate specified byte value. The byte_spec parameter represents the aligned byte to compare (0 is right-most byte, 3 is left most byte)." [Specification, page 13, line 30, to page 14, line 1]

Another inventive feature of Appellant's claim 14 is that the branch instruction causes the processor to determine whether the byte in the register is equal or not equal to a specified byte value contained in the branch instruction. "BR=BYTE, BR!=BYTE. This branch instruction branches to an instruction at the specified label if a specified byte in a longword matches or mismatches the byte_compare_value." [Specification, page 13, lines 5-7] "The ALU condition codes are set by subtracting the specified byte value from the specified register byte. If the values match, the specified branch is taken." [Specification, page 14, lines 1-3]

A further inventive feature of Appellant's claim 14 is that the branch instruction causes the processor to perform a branching operation specified by the branch instruction based on the specified byte being equal or not equal to the byte in the register. "In addition, the microengines also support a branch instruction that branches on a byte being equal or not equal to a specified byte." [Specification, page 13, lines 2-3] "BR=BYTE, BR!=BYTE. This branch instruction branches to an instruction at the specified label if a specified byte in a longword matches or mismatches the byte_compare_value." [Specification, page 13, lines 5-7]

Claim 17

Claim 17 recites another aspect of the invention. Claim 17 is a processor that comprises a register stack, and an arithmetic logic unit coupled to the register stack and a program control store that stores a branch instruction. "Referring to FIG. 1, a communication system 10 includes a processor 12." [Specification, page 1, line 25] "The hardware-based multithreaded processor 12 also includes a plurality of function microengines 22a-22f." [Specification, page 2, lines 14-15] FIG. 3 is a block diagram of a microengine used in the processor.

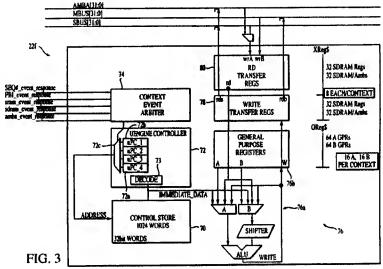

FIG. 3

"Referring to FIG. 3, an exemplary microengine 22f includes a control store 70 that includes a RAM which stores a microprogram." [Specification, page 4, lines 16-17] "The microengine 22f also includes an execution box (EBOX) data path 76 that includes an arithmetic logic unit 76a and general purpose register set 76b." [Specification, page 5, lines 6-12] "Referring to FIG. 5, the microengines support various branch instructions such as those that branch on condition codes." [Specification, page 13, lines 1-2]

Inventive features of Appellant's claim 17 also include the branch operation causing the processor to fetch a byte, specified by the branch instruction, stored in a register, specified by the branch instruction. "This branch instruction branches to an instruction at the specified label if a specified byte in a longword matches or mismatches the byte_compare_value." [Specification, page 13, lines 6-7] "Example: br!=byte[reg, byte_spec, byte_compare_value, label#], defer[3].

This instruction represents an instruction that compares an aligned byte of a register operand to an immediate specified byte value. The byte_spec parameter represents the aligned byte to compare (0 is right-most byte, 3 is left most byte)." [Specification, page 13, line 30, to page 14, line 1]

Another inventive feature of Appellant's claim 17 is that the branch instruction causes the processor to determine whether the byte in the register is equal or not equal to a specified byte value contained in the branch instruction. "BR=BYTE, BR!=BYTE. This branch instruction branches to an instruction at the specified label if a specified byte in a longword matches or mismatches the byte_compare_value." [Specification, page 13, lines 5-7]

A further inventive feature of Appellant's claim 17 is that the branch instruction causes the processor to perform a branching operation specified by the branch instruction based on the specified byte being equal or not equal to the byte in the register. "In addition, the microengines also support a branch instruction that branches on a byte being equal or not equal to a specified byte." [Specification, page 13, lines 2-3] "BR=BYTE, BR!=BYTE. This branch instruction branches to an instruction at the specified label if a specified byte in a longword matches or mismatches the byte_compare_value." [Specification, page 13, lines 5-7]

Claim 20

Claim 20 recites another aspect of the invention. Claim 20 is a method of operating a processor. "Referring to FIG. 1, a communication system 10 includes a processor 12." [Specification, page 1, line 25]

Inventive features of Appellant's claim 20 include executing a branch instruction. "BR=BYTE, BR!=BYTE. This branch instruction branches to an instruction at the specified label if a specified byte in a longword matches or mismatches the byte_compare_value." [Specification, page 13, lines 5-7]

Another inventive feature of Appellant's claim 20 includes fetching a byte, specified by the branch instruction, stored in a register, specified by the branch instruction. "This branch instruction branches to an instruction at the specified label if a specified byte in a longword matches or mismatches the byte_compare_value." [Specification, page 13, lines 6-7] "Example:

br!=byte[reg, byte_spec, byte_compare_value, label#], defer[3]. This instruction represents an instruction that compares an aligned byte of a register operand to an immediate specified byte value. The byte_spec parameter represents the aligned byte to compare (0 is right-most byte, 3 is left most byte)" [Specification, page 13, line 30, to page 14, line 1] Thus, to compare the aligned byte of a register, that byte has to be fetched.

A further inventive feature of Appellant's claim 20 is determining whether the byte in the register is equal or not equal to a specified byte value contained in the branch instruction. "BR=BYTE, BR!=BYTE. This branch instruction branches to an instruction at the specified label if a specified byte in a longword matches or mismatches the byte_compare_value." [Specification, page 13, lines 5-7]

Yet another inventive feature of Appellant's claim 20 is performing a branching operation specified by the branch instruction based on the specified byte being equal or not equal to the byte in the register. "In addition, the microengines also support a branch instruction that branches on a byte being equal or not equal to a specified byte." [Specification, page 13, lines 2-3] "BR=BYTE, BR!=BYTE. This branch instruction branches to an instruction at the specified label if a specified byte in a longword matches or mismatches the byte_compare_value." [Specification, page 13, lines 5-7]

(vi.) Grounds of Rejection to be Reviewed on Appeal

Claims 1-3, 6, 10, 13-21 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over U.S. Patent No. 4,724,521 to Carron et al, in view of U.S. Patent No. 5,802,373 to Yates and U.S. Patent No. 5,748,950 to White et al.

Claims 4 and 5 stand rejected under 35 U.S.C. 103(a) as being unpatentable over Carron, White and Yates, in view of U.S. Patent No. 5,898,866 to Atkins et al.

Claims 7, 8, and 11 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Carron, White and Yates, in view of U.S. Patent No. 4,742,151 to Bruckert et al.

Claim 9 stands rejected under 35 U.S.C. § 103(a) as being unpatentable over Carron, White and Yates, in view of U.S. Patent No. 5,202,972 Gusefski et al.

Claim 12 stands rejected under 35 U.S.C. § 103(a) as being unpatentable over Carron, White and Yates, in view of U.S. Patent No. 6,139,199 to Rodriguez.

(vii.) Argument

Obviousness

"It is well established that the burden is on the PTO to establish a prima facie showing of obviousness, *In re Fritsch*, 972 F.2d. 1260, 23 U.S.P.Q.2d 1780 (C.C.P.A., 1972)."

"To establish a prima facie case of obviousness, three basic criteria must be met. First, there must be some suggestion or motivation, either in the references themselves or in the knowledge generally available to one of ordinary skill in the art, to modify the reference or to combine reference teachings. Second, there must be a reasonable expectation of success. Finally, the prior art reference (or references when combined) must teach or suggest all the claim limitations." (MPEP 2143)

"If proposed modification would render the prior art invention being modified unsatisfactory for its intended purpose, then there is no suggestion or motivation to make the proposed modification. *In re Gordon*, 733 F.2d 900, 221 USPQ 1125 (Fed. Cir. 1984)."

"If the proposed modification or combination of the prior art would change the principle of operation of the prior art invention being modified, then the teachings of the references are not sufficient to render the claims prima facie obvious. *In re Ratti*, 270 F.2d 810, 123 USPQ 349 (CCPA 1959)."

"It is well established that there must be some logical reason apparent from the evidence or record to justify combination or modification of references. *In re Regal*, 526 F.2d 1399 188, U.S.P.Q.2d 136 (C.C.P.A. 1975). In addition, even if all of the elements of claims are disclosed in various prior art references, the claimed invention taken as a whole cannot be said to be obvious without some reason given in the prior art why one of ordinary skill in the art would have been prompted to combine the teachings of the references to arrive at the claimed invention. Id. Even if the cited references show the various elements suggested by the Examiner in order to support a conclusion that it would have been obvious to combine the cited references, the references must either expressly or impliedly suggest the claimed combination or the Examiner

must present a convincing line of reasoning as to why one skilled in the art would have found the claimed invention obvious in light of the teachings of the references. *Ex Parte Clapp*, 227 U.S.P.Q.2d 972, 973 (Board. Pat. App. & Inf. 985)."

"The mere fact that the prior art could be so modified would not have made the modification obvious unless the prior art suggested the desirability of the modification." *In re Gordon*, 221 U.S.P.Q. 1125, 1127 (Fed. Cir. 1984).

> Although the Commissioner suggests that [the structure in the primary prior art reference] could readily be modified to form the [claimed] structure, "[t]he mere fact that the prior art could be so modified would not have made the modification obvious unless the prior art suggested the desirability of the modification." *In re Laskowski*, 10 U.S.P.Q. 2d 1397, 1398 (Fed. Cir. 1989).

"The claimed invention must be considered as a whole, and the question is whether there is something in the prior art as a whole to suggest the desirability, and thus the obviousness, of making the combination." *Lindemann Maschinenfabrik GMBH v. American Hoist & Derrick*, 221 U.S.P.Q. 481, 488 (Fed. Cir. 1984).

> Obviousness cannot be established by combining the teachings of the prior art to produce the claimed invention, absent some teaching or suggestion supporting the combination. Under Section 103, teachings of references can be combined only if there is some suggestion or incentive to do so. *ACS Hospital Systems, Inc. v. Montefiore Hospital*, 221 U.S.P.Q. 929, 933 (Fed. Cir. 1984) (emphasis in original, footnotes omitted).

"The critical inquiry is whether 'there is something in the prior art as a whole to suggest the desirability, and thus the obviousness, of making the combination.'" *Fromson v. Advance Offset Plate, Inc.*, 225 U.S.P.Q. 26, 31 (Fed. Cir. 1985).

(1) Claims 1-3, 6, 10, 13-21 are patentable over Carron, in view of Yates and White

Claims 1, 2, 3, 6, 10, 12, 13, and 14-21

For the purposes of this appeal only, claims 1, 2, 3, 6, 10, 13 and 14-21 stand or fall together. Claim 1 is representative of this group of claims.

Claim 1 is directed to a method for operating a processor that includes the features of "executing a branch instruction ... based on a byte, specified by the branch instruction, in a register, specified by the branch instruction, being equal or not equal to a byte value specified by the branch instruction." The Examiner combined three references, namely Carron, White and Yates, to reject independent claim 1.

Appellant contends that the examiner has failed to show all of the features of claim 1 through the alleged combination of references. In particular the examiner has failed to show at least the feature of "a byte value specified by the branch instruction." Assuming for the moment that Carron and White teach the features the examiner relies on, there are no teaching however in those references of ... a byte value specified by the branch instruction. The examiner acknowledges this and uses Yates. Yates is directed to a translator program that translates a non-native code into a native code such that code compiled for one architecture can run on a completely different architecture.

Yates apparently teaches a compare instruction that specifies a value "Source instruction 884a (CMPB)" to perform a byte compare of in the register AL to the constant 3. Yates however uses a separate instruction "Instruction 884b (BNE)" that performs a branch based on the results of the previous instruction (FIG. 65B, and col. 77, lines 29-33). In no sense, therefore, would any combination of Carron, White and Yates suggest executing a branch instruction ... based on a byte, specified by the branch instruction, in a register, specified by the branch instruction, being equal or not equal to a byte value specified by the branch instruction.

Appellant further contends that notwithstanding the failure of the combined references to teach all the features of independent claim 1, for the reasons that follow, Appellant contends that there exists no motivation for combining these three references.

Combining White and/or Yates with Carron Would Render Carron Unsatisfactory for Its Intended Purpose and/or Change its Principle of Operation

Carron describes a method for operating a local terminal that includes executing a set of instructions by a central processor unit in a prearranged manner to accomplish a specific task. As Carron explains:

> **The method of this invention provides a practicable approach to downloading an entire custom application program because the size of the program code required to be downloaded is reduced by a factor of three or four by avoiding the requirement of downloading all of the actual machine code instructions of the application program.**
>
> **Instead, the method of this invention is based on storing in read only memory circuits within the local terminal a number of general purpose operation routines which comprise instructions to be executed by the central processor unit to accomplish a particular program task. Each of these general purpose operation routines is associated with a defined command which, in its object code version, includes an operation code. The object code version of the commands associated with the general purpose operation routines has a code length substantially shorter than the code length of the operation routine. In accordance with the method of this invention, the local terminal also has a program established in its read only memory system for interpreting the operation code to access the associated general purpose operation routine for execution by the central processor unit. The object code version of the application program in the form of a sequence of commands has a code size which is several times smaller than the compiled code size would be for an entire application program which could be directly executed after downloading.**
>
> **More specifically, the method of this invention includes a first step of establishing a set of general purpose operation routines to be executed by the local computer system. Each of these general purpose operation routines comprises a set of instructions for execution by the central processor unit in a prearranged manner to accomplish a specific task. The next step is to store the set of general purpose operation routines in the read only memory of the local terminal so that they may be accessed for execution by the central processor unit. The storage locations of these general purpose operation routines will be arranged and noted so that the routines can be addressed.**
>
> **A following step is to define a set of commands, each of which is associated with a specific one of the general purpose operation routines. Each of the commands includes at least an operation code relating the command to its associated general purpose operation routine. Each of the commands is defined such that it has an associated command code length substantially less than the code length of the associated general purpose operation routine. Preferably, for convenience of writing programs, each command is defined with a high level programming syntax in which the command is**

> **represented by a series of word forms in abbreviated notation which have a recognizable association with the task that the associated general purpose operation routine will perform when it is executed by the central processor unit in the local terminal. This high level form of the command is then compiled and assembled to produce the operation code which relates to the associated general purpose operation routine in a manner which is intelligible to the interpreter program in the local terminal. (emphasis added, col. 7, line 52, to col. 8, line 47).**

Carron defines commands that correspond to routines stored in memory. For the sake of argument one can view these commands as sets of instructions. For example, one of those commands, listed in Carron's "SECTION FOUR: ALPHABETIC LISTING OF COMMANDS", is the COMP_BYTE command (col. 134, lines 5-26). However, Carron's COMP_BYTE, much like Carron's other commands, is not a processor-executable instruction, but rather is a command (or instruction) stored in the memory of a terminal executing the command. Carron explicitly states that it seeks to avoid having to download instructions at remote terminals as an objective of its method (see col. 7, lines 53-58).

Carron further explains that one of the advantages of its programming methodology is that:

> **It should be apparent from the above description that the method of this invention makes it possible for the programmer to create application programs with commands in a source level language that are particularly suited to the type of application for the terminals that are to be programmed. The commands are compiled into compressed operation codes and parameters which reduce the size of the code which must be downloaded to a practicable size. The methods of this invention provide for parallel execution of APMs which greatly expands the facility with which the programmer can implement the execution of the program tasks in the local terminal without wasting time for inputs to arrive. This increases the efficiency of use of the central processor unit within the terminal. (Carron, col. 105, lines 49-63)**

Appellant contends that Carron's commands correspond to a high-level programming language instructions that a programmer may use to prepare application programs that include sets of commands that can be executed on different terminals (i.e., presumably having different processor architectures). Accordingly, commands, such as Carron's COMP_BYTE command, cannot include instructions that are unique to a particular processor architecture because such

commands could not be executed on some other processor architectures. Carron's COMP_BYTE command also cannot include architecture-specific operands such as registers.

In contrast, White describes an optimized compare-and branch (COBR) instruction for execution in a RISC-type processor (Abstract). As shown in FIG. 3, White's COBR instruction includes the opcode, the operands SRC1 and SRC2, and the displacement field (i.e., the displacement to the target branching address). White explains that:

> **When a COBR instruction is executed, the microprocessor compares the source 2 and source 1 operands provided with the instruction and sets a condition code in a register located in the instruction sequencer according to the comparison results. Subsequently, a portion of the instruction's opcode is compared against the actual condition code. If the comparison results in a match, then the processor branches to an instruction specified by the displacement value propagated with the COBR instruction. Otherwise, the processor goes to the next sequential instruction. Mechanisms for comparing condition codes to a portion of an instruction's opcode are well known for determining branch decisions and will not be described more fully herein. (Col. 5, lines 44-56)**

With respect to the operands used by White's various instructions, including the COBR instruction, White further explains:

> **Within the processor illustrated in FIG. 2, all operations take place at the register level. Source operands specify either a global register, a local register or a constant value as instruction operands. The functional units are coupled to the register file 30 via three independent 32-bit buses. These are identified as source 1 (SRC1), source 2 (SRC2) and the destination (DEST) buses. In alternative embodiments of the present invention, a wider single bus, or smaller multiplexed common bus may be utilized (or various combinations of separate buses) for communicating between the register file 30 and the various functional units. (col. 5, lines 14-24)**

Thus, White's COBR instruction is a machine level instruction that performs a comparison operation using registers, and is configured to be executed on the particular processing environment shown in FIG. 2.

Yates describes a computer system for executing a binary image conversion which translates instructions from an instruction set of a first non-native system to a second native computer system (Abstract). Amongst the operations that Yates' system performs is condition code processing (described at col. 76, line 11 to col. 78, line 34). One such condition code processing sequence is shown in Yates' FIG. 65B. As explained by Yates, "[s]ource instruction

884a performs a byte compare of register AL to the constant 3. Instruction 884b performs a branch if the value contained in the register AL is not equal to 3" (col. 77, lines 29-32).

Yates translates the received, non-native instruction sequences into instruction sequences that can be executed on the native microprocessor. Thus, unlike Carron's commands, Yates instruction sequences, including the instruction sequences 884a and 884b shown in FIG. 65B, are machine-level instructions configured for executing in a particular processing environment.

Because Carron requires that commands, including the BYTE_COMP command, are provided in a compact high-level language that does not refer to architecture-specific operands, such as registers, and that could potentially be compatible with different processors, to modify Carron's high-level commands by combining it with White's and/or Yates machine-level instructions would render Carron unsatisfactory for its intended purpose and/or change Carron's principle of operation. Indeed, not only are Carron's high-level commands incompatible with White's and Yates' machine-level instructions, but also use of White's and Yates machine-instructions would preclude Carron's commands from executing on different types of processors, and could slow down downloading of application programs to Carron's remote terminals.

Because combining White and/or Yates with Carron would render Carron unsatisfactory for its intended purpose and/or would change Carron's principle of operation, no motivation for combining Carron with Yates and/or White exists. The examiner has thus failed to establish a *prima facie* case of obviousness with respect to independent claim 1, as well as with respect to independent claims 14, 17 and 20. For this reason alone independent claim 1 is, therefore, patentable over the cited art.

The References Teach Away from Their Combination

It is improper to combine references where the references teach away from their combination. *In re Grasselli,* 713 F.2d 731, 743, 218 USPQ 769, 779 (Fed. Cir. 1983) (The claimed catalyst which contained both iron and an alkali metal was not suggested by the combination of a reference which taught the interchangeability of antimony and alkali metal with the same beneficial result, combined with a reference expressly excluding antimony from, and adding iron to, a catalyst.). [MPEP,§2145.X.D.2]

As explained above, Carron's high-level commands are intended to avoid using machine-level instructions that identify architecture-specific operands, such as registers. In contrast, White provides that: "[w]ithin the processor illustrated in FIG. 2, all operations take place at the register level. Source operands specify a global register, a local register or a constant value as instruction operands." Thus, White's COBR instruction is implemented using registers. Similarly, Yates explains that, "[s]ource instruction 884a performs a byte compare of register AL to the constant 3. Instruction 884b performs a branch if the value contained in the register AL is not equal to 3" (col. 77, lines 29-32). Thus, Yates disclosed instructions are also machine-level instructions that identify architecture-specific operands.

Because Carron's commands, including the COMP_BYTE command, are designed to avoid directly referring or using registers and other architecture-specific resources/operands, whereas White and Yates implement various features that the examiner cannot find in Carron COMP_BYTE command, but using registers, both White and Yates teach away from Carron and are indeed incompatible with Carron. Accordingly, Appellant submits that the White and Yates references cannot be combined with the Carron reference.

Therefore, for this reason also, Appellant submits that there is no motivation to combine Carron with White and Yates. The Examiner has thus failed to establish a *prima facie* case of obviousness with respect to independent claims 1, as well as with respect to independent claims 14, 17 and 20.

Because a *prima facie* case of obviousness has not been established with respect to independent claims 1, 14, 17 and 20, Appellant submits that independent claims 1, 14, 17 and 20, and the claims that depend from them, are patentable over the cited art.

(2) Claims 4 and 5 are patentable over Carron, White and Yates, in view of Atkins

Claims 4 and 5

For the purpose of this appeal only, claims 4 and 5 may be treated as standing or falling together. Claim 4 is representative of this group

Claim 4 adds the distinct feature that the branch instruction comprises an optional token that is set by a programmer and specifies a number of instructions *i* to execute following execution of the branch instruction and before performing the branch operation. As explained in Appellant's Specification:

> **The instruction also can include an optional_token. In this example the optional token can be a defer one value which will execute one instruction following this branch instruction before performing the branch operation. The optional token can alternatively be a defer two instructions which is allowed with the br!=byte instruction and executes the two instructions following this branch instruction before performing the branch operation. The optional token can alternatively be a defer three instructions which is allowed with the br!=byte instruction and which causes the processor to execute the three instructions following this branch instruction before performing the branch operation. [Specification, page 13, lines 21-29]**

Thus, the optional token specifies the number of instruction that are to be executed after the branch instruction was executed, but before the branching operation was performed.

The Examiner admits that Carron does not teach this feature, but contends that Atkins in Col. 2, lines 28-31, and Col. 10, lines 21-23, shows this feature. Atkins describes:

> **A specialized instruction (i.e., SETLOOP) is executed to initialize the loop operation. This instruction has a length field which specifies the number of instructions within the loop. This number is loaded into a counter which is decremented synchronously with the execution of each instruction of the loop, thereby maintaining an instantaneous indication of the position within the loop. When the loop is repeated, the length value is again loaded into the counter. This counter permits prefetching of branch instructions as desired.**
>
> **A second field in SETLOOP instruction contains the number of times which the loop is to be executed. This number is loaded into a second counter which is decremented with each execution of the loop. The value of this second counter is used to determine which instruction to request (i.e., top-of-loop or outside-of-loop) at the end of each iteration.**
>
> **A register is used to store the address of the top-of-loop instruction. This ensures that the instruction may be fetched to begin the next iteration of the loop during each previous iteration. Push-Pop stacks may be used to back up this register and each of the two counters to permit loop nesting. (Atkins, col. 2, lines 28-49)**

Atkins further explains:

> **The SETLOOP instruction initializes the loop control hardware REG N serves as the counter to count the number of iterations to execute the loop. It is decremented with each iteration to control the branch at the bottom of the loop. BOT is the number of instructions within the loop (i.e., two). This**

number is entered into the field by the compiler or other system software. It is loaded into a special register which is decremented as each loop instruction is executed. The register is reloaded with BOT after each branch to the beginning of the loop for a new iteration. In this manner, the instantaneous position within the loop is always known. (Atkins, col. 9, line 66, to col. 10, line 9)

The Examiner's characterization of Atkins is incorrect. Atkins' so called "SETLOOP" instruction includes a parameter specifying the number of instructions within a loop. Atkins maintains a counter using a register that is decremented as each loop instruction is executed. That counter indicates how many instructions remain to be executed before the end of the loop and thus, as Atkins states "[t]his counter permits prefetching of branch instructions as desired." Therefore, the parameter specified for the SETLOOP instruction has no relevance to the claim feature, but rather indicates the number of instructions before a branch instruction. In contrast, Appellant's claimed feature specifies the number of instructions to execute after the branch instruction. Accordingly, Atkins does not disclose or suggest at least "wherein the branch instruction comprises: an optional token that is set by a programmer and specifies a number i of instructions to execute following the branch instruction before performing the branch operation."

Although the Examiner implicitly acknowledges that White and Yates do not disclose the feature recited in Appellant's claim 4, Appellant notes that indeed neither of these two references discloses at least "wherein the branch instruction comprises: an optional token that is set by a programmer and specifies a number i of instructions to execute following the branch instruction before performing the branch operation."

Because none of the references cited by the Examiner to reject Appellant's claim 4 discloses or suggests, alone or in combination, at least the feature of "wherein the branch instruction comprises: an optional token that is set by a programmer and specifies a number i of instructions to execute following the branch instruction before performing the branch operation," Appellant's claim 4 is patentable over the cited art.

(3) Claims 7, 8, and 11 are patentable over Carron, White and Yates, in view of Bruckert et al.

Claims 7 and 11

For the purposes of this appeal only, claims 7 and 11 may be treated as standing or falling together. Claim 7 is representative of this group of claims.

Claim 7 recites the additional distinct feature that the branch instruction comprises an optional token that is set by the programmer and which specifies a guess_branch prefetch for the instruction for the "branch taken" condition rather than the sequential instruction.

The Examiner admits that Carron does not teach this feature, but contends that Bruckert in Col. 1, lines 22-29 shows this feature. Appellant respectfully submits that the Examiner's characterization of Bruckert is incorrect. Bruckert describes:

> **The invention enables the processor to determine if an instruction is a conditional branch instruction, in which a determination is made as to whether a branch in an instruction stream should or should not be taken depending on the results of prior processing, and to prefetch both the instructions in the "branch taken" instruction stream as well as from the "branch not taken" instruction stream. (Bruckert, col. 1, lines 22-29)**

Thus, Bruckert merely describes a processor that fetches two instructions when the processor encounters a branch instruction: one instruction corresponding to the "branch taken" eventuality (i.e., should the evaluation of the branch condition result in a branch taken operation), as well as the instruction corresponding to the "branch not taken" condition (i.e., should the evaluation of the branch condition result in the branch not being taken.). Neither of these eventualities correspond to the claimed branch guess token. Because both "branch taken" and "branch not taken" instruction sequences are retrieved, Bruckert would have no need to include a token that would indicate which instruction should be retrieved following the execution of a branch instruction. Bruckert neither discloses nor suggests at least "the branch instruction comprises: an optional token that is set by a programmer and which specifies a guess_branch prefetch for the instruction for the "branch taken" condition rather than the next sequential instruction," as required by Appellant's claim 7.

Although the Examiner implicitly acknowledges that White and Yates do not disclose the feature recited in Appellant's claim 7, Appellant further notes that indeed neither of these two references discloses at least "the branch instruction comprises: an optional token that is set by a

programmer and which specifies a guess_branch prefetch for the instruction for the "branch taken" condition rather than the next sequential instruction."

Because none of the reference cited by the Examiner to reject Appellant's claim 7 discloses or suggests, alone or in combination, at least the feature of "the branch instruction comprises: an optional token that is set by a programmer and which specifies a guess_branch prefetch for the instruction for the "branch taken" condition rather than the next sequential instruction," Appellant's claim 7 is therefore patentable over the cited art.

Claim 8

Appellant's claim 8 recites the additional distinct feature that the branch instruction further includes an optional token that is set by the programmer and specifies a number of instructions to execute following the branch instruction before performing the branch operation, as well as the distinctive feature that the instruction also includes a second optional token that is specified by the programmer and which specifies a guess_branch prefetch for the instruction for the "branch taken" condition rather than the next sequential instruction.

The Examiner argued that:

> 26. As per claims 7 and 8, Carron taught of instructions to execute following the branch instruction before performing the branch operation (Col 134, lines 8-26). However, Carron does not teach the method wherein the branch instruction comprises: an optional token that is set by a programmer and which specifies a guess_branch prefetch for the instruction for the "branch taken" condition rather than the next sequential instruction.
>
> 27. Bruckert teaches of executing instructions where a determination is made as to whether a branch should or should not be taken, and teaches of prefetching "branch taken " instructions (Col 1, lines 22-29). (Final action, Page 7)

Appellant disagrees with the Examiner's contentions.

With respect to the Examiner's contention that "Carron taught of instruction to execute following the branch instructions before performing the branch operation", Appellant notes that the Examiner has already conceded, in relation to Appellant's claims 4 and 5 that recite a similar feature, that:

22. As per claims 4 and 5, Carron does not teach the method of claim 1 wherein the branch instruction comprises: an optional token that is set by a programmer and specifies a number i of instructions to execute following the branch instruction before performing the branch operation where the number of instruction before performing the branch operation where the number of instructions can be specified as one, two, or three. (Final Action, page 6)

Indeed, Carron's description of the COMP_BYTE command provides:

COMP_BYTE

Compare byte at specified position in designated buffer with the value stored in the specified variable. Branch if test passes. Continue with the next command if the test fails. Table 3 contains a list of the comparison operators.

COMP_BYTE	**<buffer> <position> <operator> <variable> <Addr>**
Opcode Group:	**1**
Opcode Number:	**24**
Source Syntax:	**BXCVVA**
Object Seq.:	**2-4-1-3-5**
Object Syntax:	**XVBXA**
Entry Point:	**#CMBYT**
Max Opcode #:	**18**
<buffer>	**Destination buffer.**
<position>	**Position in the destination buffer to be be compared. First char = 1.**
<operator>	**Type of comparison requested. (Table 3)**
<variable>	**Address of eight bit variable.**
<Addr>	**Branch address if test passes.**

At no point does Carron disclose or suggest an optional token that specifies a number *i* of instructions to execute following a branch operation. Nor does Carron disclose or suggest anywhere that its COMP_BYTE command performs any type of branch defer operation that causes the execution of additional instructions (or commands) following the branch instruction and before the branch operation is performed.

As Appellant explained with respect to claim 4, Atkins (the reference relied upon by the Examiner to reject claim 4) also does not disclose this feature.

Accordingly, the prior art cited by the Examiner does not disclose or suggest the feature of "an optional token that is set by a programmer and specifies a number i of instructions to execute following the branch instruction before performing the branch operation," required by Appellant's claim 8.

As for the feature of the optional token pertaining to the guess_branch prefetch, for reasons similar to those provided with respect to claim 7, this feature is not disclosed or suggested by Bruckert.

The Examiner did not explicitly rely on either the White or the Yates references to reject any of the features recited in claim 8. Appellant, therefore, considers that the Examiner implicitly acknowledges that neither of these references discloses these features. Appellant further notes that neither of these two references discloses at least "an optional token that is set by a programmer and specifies a number i of instructions to execute following the branch instruction before performing the branch operation, and a second optional token that is set by a programmer and which specifies a guess_branch prefetch for the instruction for the "branch taken" condition rather than the next sequential instruction."

Because none of the references cited by the examiner discloses or suggests, alone or in combination at least "an optional token that is set by a programmer and specifies a number i of instructions to execute following the branch instruction before performing the branch operation, and a second optional token that is set by a programmer and which specifies a guess_branch prefetch for the instruction for the "branch taken" condition rather than the next sequential instruction," claim 8 is therefore patentable over the cited art.

(4) Claim 9 is patentable over Carron, White and Yates, and Gusefski

Claim 9

Claim 9 is allowable over the combination of references at least for the reasons discussed in claim 1, and Gusefski does not cure the deficiencies pointed out by combining Carron with White and Yates .

(5) Claim 12 is patentable over Carron, White and Yates, in view of Rodriguez

Claim 12 is allowable over the combination of references at least for the reasons discussed in claim 1, and Rodriguez does not cure the deficiencies pointed out by combining Carron with White and Yates .

Conclusion

Appellant submits, therefore, that Claims 1-21 are allowable over the cited art. Therefore, the Examiner erred in rejecting Appellant's claims and should be reversed.

Respectfully submitted,

Date: Sept. 14, 2006

Ido Rabinovitch
Reg. No. L0080

PTO Customer No. 26161
Fish & Richardson P.C.
Telephone: (617) 542-5070
Facsimile: (617) 542-8906

Appendix of Claims

1. A method of operating a processor comprising:

executing a branch instruction that causes a processor to branch from executing a first sequential series of instructions to a different sequential series of instructions based on a byte, specified by the branch instruction, in a register, specified by the branch instruction, being equal or not equal to a byte value specified by the branch instruction.

2. The method of claim 1 wherein the branch is to an instruction at a specified label.

3. The method of claim 1 wherein the branch instruction comprises:

a bit_postion field that specifies the byte in a longword contained in the register.

4. The method of claim 1 wherein the branch instruction comprises:

an optional token that is set by a programmer and specifies a number i of instructions to execute following the branch instruction before performing the branch operation.

5. The method of claim 1 wherein the branch instruction comprises:

an optional token that is set by a programmer and specifies a number i of instructions to execute following the branch instruction before performing the branch operation where the number of instructions can be specified as one, two or three.

6. The method of claim 1 wherein the register is a context-relative transfer register or a general-purpose register that holds the operand.

7. The method of claim 1 wherein the branch instruction comprises:

an optional token that is set by a programmer and which specifies a guess_branch prefetch for the instruction for the "branch taken" condition rather than the next sequential instruction.

8. The method of claim 1 wherein the branch instruction comprises:

an optional token that is set by a programmer and specifies a number i of instructions to execute following the branch instruction before performing the branch operation; and

a second optional token that is set by a programmer and which specifies a guess_branch prefetch for the instruction for the "branch taken" condition rather than the next sequential instruction.

9. The method of claim 1 wherein the branch instruction allows a programmer to specify which bit of the register to use to determine the branch operation.

10. The method of claim 1 wherein the branch instructions allows branches to occur based on evaluation of a byte that is in a data path of a processor.

11. The method of claim 1 wherein the branch instruction branches on the byte matching the byte value and wherein the instruction prefetches the instruction for the "branch taken" condition.

12. The method of claim 1 wherein the branch instruction branches on the byte not matching the byte value and wherein the instruction prefetches the next sequential instruction.

13. The method of claim 1 wherein the branch instruction includes a Byte_spec Number that specifies the byte in the register to be compared with byte_compare_value.

14. A computer program product residing on a computer readable medium comprising instructions, including a branch instruction that causes a processor to:

fetch a byte, specified by the branch instruction, stored in a register, specified by the branch instruction;

determine whether the byte in the register is equal or not equal to a specified byte value contained in the branch instruction; and

perform a branching operation specified by the branch instruction based on the specified byte being equal or not equal to the byte in the register.

15. The product of claim 14 wherein the branch is to an instruction at a specified label.

16. The product of claim 14 wherein the branch instruction comprises:

a bit_postion field that specifies the byte in a longword contained in the register.

17. A processor comprises:

a register stack;

an arithmetic logic unit coupled to the register stack and a program control store that stores a branch instruction that causes the processor to:

fetch a byte, specified by the branch instruction, stored in a register, specified by the branch instruction;

determine whether the byte in the register is equal or not equal to a specified byte value contained in the branch instruction; and

perform a branching operation specified by the branch instruction based on the specified byte being equal or not equal to the byte in the register.

18. The processor of claim 17 wherein the branch instruction that causes the processor to perform the branching operation causes the processor to branch to an instruction at a specified label.

19. The processor of claim 17 wherein a bit_postion field in the branch instruction specifies the byte in a longword contained in the register.

20. A method of operating a processor comprises:

executing a branch instruction by:

fetching a byte, specified by the branch instruction, stored in a register, specified by the branch instruction;

determining whether the byte in the register is equal or not equal to a specified byte value contained in the branch instruction; and

performing a branching operation specified by the branch instruction based on the specified byte being equal or not equal to the byte in the register.

21. The method of claim 20 wherein performing the branch includes branching to an instruction at a specified label.

EVIDENCE

APPENDIX

None

RELATED PROCEEDINGS
APPENDIX

None